Hardware Atomicity for Reliable Software Speculation

Naveen Neelakantam†‡, Ravi Rajwar‡, Suresh Srinivas‡, Uma Srinivasan‡, and Craig Zilles†

University of Illinois at Urbana-Champaign† and Intel Corporation‡
[neelakan, zilles]@uiuc.edu, [ravi.rajwar, suresh.srinivas, uma.srinivasan]@intel.com

ABSTRACT

Speculative compiler optimizations are effective in improving both single-thread performance and reducing power consumption, but their implementation introduces significant complexity, which can limit their adoption, limit their optimization scope, and negatively impact the reliability of the compilers that implement them. To eliminate much of this complexity, as well as increase the effectiveness of these optimizations, we propose that microprocessors provide architecturally-visible hardware primitives for atomic execution. These primitives provide to the compiler the ability to optimize the program's hot path in isolation, allowing the use of non-speculative formulations of optimization passes to perform speculative optimizations. Atomic execution guarantees that if a speculation invariant does not hold, the speculative updates are discarded, the register state is restored, and control is transferred to a non-speculative version of the code, thereby relieving the compiler from the responsibility of generating compensation code.

*We demonstrate the benefit of hardware atomicity in the context of a Java virtual machine. We find incorporating the notion of atomic regions into an existing compiler intermediate representation to be natural, requiring roughly 3,000 lines of code (∼3% of a JVM's optimizing compiler), most of which were for region formation. Its incorporation creates new opportunities for existing optimization passes, as well as greatly simplifying the implementation of additional optimizations (*e.g.*, partial inlining, partial loop unrolling, and speculative lock elision). These optimizations reduce dynamic instruction count by 11% on average and result in a 10-15% average speedup, relative to a baseline compiler with a similar degree of inlining.*

Categories and Subject Descriptors: D.3.4 [**Software**]: Programming Languages–*Processors:Compilers, Optimization*, C.0 [**Computer Systems Organization**]: General–*Hardware/software interfaces*

General Terms: Performance

Keywords: Atomicity, Checkpoint, Isolation, Java, Optimization, Speculation

1. INTRODUCTION

In his landmark paper "Compilers and Computer Architecture," William Wulf identifies three principles (regularity, orthogonality, and composability) that instruction sets should adhere to in order to simplify compiler implementations, thereby improving the code quality that is practically achievable [21]. Each time these principles are not observed, an additional set of special cases must be considered during compilation in order to generate the best possible code for a given program. While architectures that ignore these principles do not, in theory, preclude the building of compilers that generate the highest performance code, in practice the quality of code suffers as many compiler implementations will be unable to justify the additional software complexity required.

Extending the principles set forth by Wulf, we see *atomic execution*—executing a region of code completely (and as if all operations in the region occurred at one instant) or not at all—as a fundamental principle for both improving the effectiveness of existing compiler optimizations and simplifying the implementation of additional compiler optimizations. Specifically, we propose that microprocessors expose atomicity as a hardware primitive to the compiler. Doing so permits the compiler to generate a *speculative version* of the code where uncommonly executed code paths are completely removed, so that they need not be considered in (and hence do not constrain) a region's optimization. If one of these pruned paths needs to be executed, the region will be aborted—reverting back to the state at the beginning of the region—and control will be transferred to a non-speculative version of the code.

Speculative optimizations are important for achieving high performance in many integer and enterprise applications, because the control flow intensive nature of these programs prevents non-speculative compiler approaches from generating efficient code. In fact, the presence of frequent control flow can be a significant inhibitor of compiler optimizations, even when a significant fraction of the control flow is strongly-biased and compilation is performed with an accurate profile, as is true for the run-time optimizers invoked in modern Java* virtual machines.

Consider, for example, the inter-procedural control flow graph of the fully-optimized code generated by a leading commercial JVM for the most frequently executed loop from Jython*(a DaCapo* benchmark [2]) shown schematically in Figure 1(a). During execution, few paths through this loop are ever executed, but the hottest of those paths executes 109 conditional branches and over 600 instructions. Our manual analysis of the hot path found that aggressive speculative optimizations can remove more than two-thirds of the instructions (Figure 1(b)), which translates into both improved performance and reduced power consumption. Traditional approaches to implementing these speculative optimizations, however, come at a significant cost in complexity, because correct execution along all of the potential paths has to be preserved by the compiler. Figure 1(c) shows the simplest possible control flow graph—having aggressively eliminated 67 branches with redundant

ISCA'07, June 9–13, 2007, San Diego, California, USA.

*Other names and brands may be claimed as the property of others.

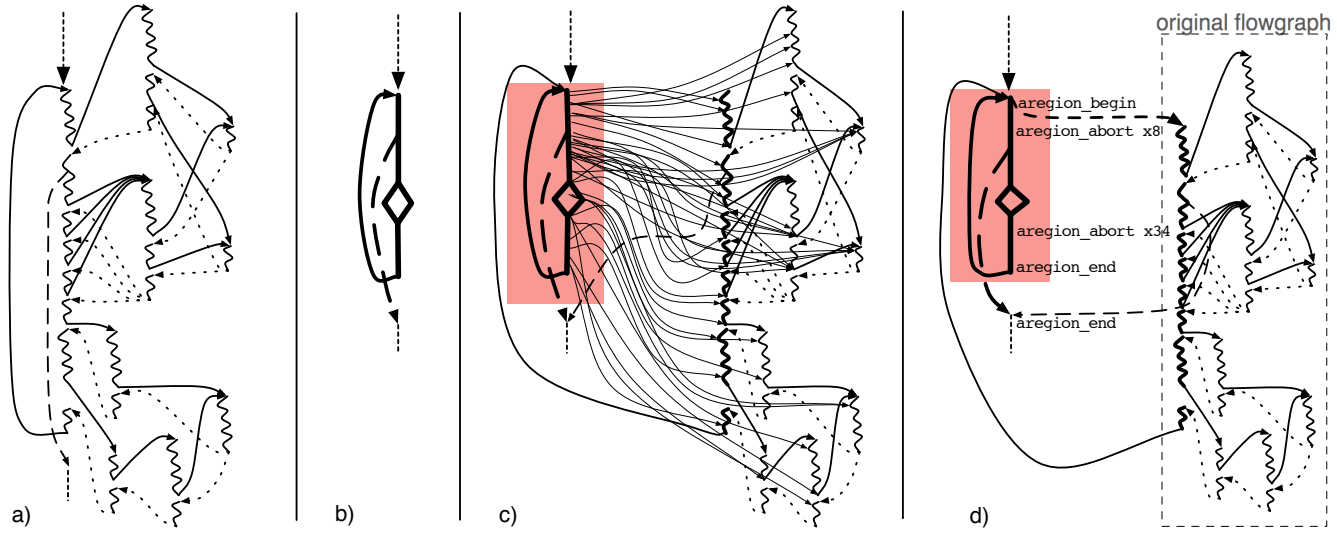

Figure 1: Complexity of Compiler Optimizations. Abstract inter-procedural control flow graph from Jython (only executed paths shown). (a) as optimized by a commercial JVM, (b) the hot path if optimized in isolation, (c) the control flow/call graph resulting from partial-inlining and superblock formation to optimize the hot paths, (d) the control flow/call graph using the proposed hardware support for atomic regions. Atomic regions enable the compiler to isolate the hot path from the cold paths for the purpose of optimization; if one of the compiler's speculations should fail, state is rolled back to the beginning of the atomic region and control is transferred to a non-speculative version of the code.

conditions—that achieves the desired optimization of the hot path. Because of the difficulty of verifying the correctness of these radical program transformations, many commercial systems do not perform speculative optimizations to this extent.

The fundamental source of complexity for the flow graph in Figure 1(c) is the compiler's inability to *isolate* the hot path from the cold path. The compiler must guarantee that any exit from the hot path, however unlikely, will generate correct results. It must provide two key assurances to fulfill this guarantee. First, the compiler must ensure that sufficient program state is kept live in the hot path such that at each exit the "precise" program state required by the cold path can be reconstructed. Second, it must maintain mappings from the optimized hot path's state to that of the cold path so that compensation code can be generated, for every exit from the hot path, to undo any hot-path specific optimizations.

Atomic execution, however, obviates the need for this complexity, as shown in Figure 1(d). The compiler merely replicates the hot code for execution in an *atomic region*; the entry to the atomic region is delimited by an instruction (`aregion_begin`) that communicates the beginning of speculative execution to the hardware, and exits from the atomic region are delimited by an instruction (`aregion_end`) that instructs the hardware to commit the region's results atomically. The compiler converts branches to cold paths into conditional abort instructions (`aregion_abort`); if an abort condition evaluates such that control should transfer to a cold path, the hardware rolls back to the state prior to the `aregion_begin`, and transfers control to the original (non-speculative) version of the code, *as if speculative execution of the hot path had never occurred*.

To summarize, atomic execution primitives simplify the implementation of speculative optimizations in three ways:

- **Hardware maintains the state necessary for recovering from speculative optimizations.** The compiler no longer needs to generate compensation code that recovers the correct program state at each exit from the hot path.
- **Hardware atomicity enables analysis and optimizations to ignore the cold paths when optimizing the hot paths.** By converting cold paths into conditional aborts, the compiler enables existing *non-speculative* analysis routines and optimizations to perform, in effect, path-qualified analysis and speculative optimizations. In comparison, previous approaches to speculative optimization require complete re-implementation of these compiler passes.
- **Hardware isolates execution from other threads.** The compiler need not worry about the multithreaded safety of optimizations within the atomic region, because memory operations in the region appear to occur atomically with respect to memory operations from other threads.

In this paper, we report on our experience exploiting hardware atomic regions to optimize single-thread performance in a Java virtual machine. Our key findings can briefly be summarized as follows:

- Hardware atomicity greatly improves the return-on-investment in implementing compiler optimizations. By using atomic regions, we enable much higher code quality for a given amount of compiler complexity. That is, hardware atomicity improves the effectiveness of an optimization or simplifies its implementation or both.
- Atomic execution primitives provide a clean abstraction for compiler implementation (Section 4). We find that incorporating atomic regions into an existing compiler internal representation (IR) required minimal re-engineering of existing IR nodes and compiler passes; we extend existing support for `try`/`catch` primitives. In addition, atomic regions can be formed very early in the compilation process, benefiting optimizations and transformations on both the high-level IR (*e.g.*, inlining and loop unrolling), as well as the low-level IR. Other systems have provided hardware support for compiler optimizations (*e.g.*, IA-64), but the abstractions provided are overly complicated and restrict their usefulness to back-end optimizations (*e.g.*, scheduling).

- The atomic region abstraction simplifies the implementation of new optimizations. For example, one author was able to produce a working implementation of partial inlining in 6 hours. Without hardware atomic regions, this is a difficult transformation to implement. When discussing the correctness of their partial inliner implementation (which did not use atomic regions), Muth and Dubray remark [15]: "The flow of control in the program resulting from partial inlining is sufficiently complex that it is no longer obvious that the resulting program is semantically equivalent to the original."
- Atomic regions enable existing non-speculative formulations of optimizations to perform transformations that typically require speculative formulations (Section 6).
- A high-performance hardware implementation is a necessity, as any overhead translates directly to lost performance (Section 6.3). Defining atomic execution primitives with a clean interface permits high-performance hardware implementations (Section 3). The compiler further facilitates a high-performance implementation by being tolerant of "best effort" implementations that abort for cases that are difficult for hardware designers (*e.g.*, data footprint overflowing the L1 cache).

In support of these findings, we evaluate the quality of code generated from our prototype compiler enhanced with atomic regions using detailed timing simulation. In our experiments, atomic region-based optimizations enabled single-thread speedups (on an aggressive superscalar processor) as high as 35%, with averages across a collection of Java benchmarks from the DaCapo benchmark suite of 10% and 15%, depending on the degree of inlining performed (Section 6). Furthermore, we find that our implementation results in an 11% average reduction in the number of micro-operations executed by the processor. We describe our experimental method for generating execution samples from Java-based workloads using different compilers in Section 5.

As these experiments represent only an initial implementation of compiler support for atomic regions, these results should not be considered as definitive or as bounding the potential of atomic regions. For example, we found that some of the optimization potential that our atomicity-enabled speculative optimizations expose in the JVM is not being effectively exploited due to limitations in the compiler's back end (Section 6.4). Clearly, atomic regions are not a panacea; a compiler will only generate code as good as its weakest pass. In addition, we find that if the abort rate is more than a few percent, atomic regions begin to hurt performance rather than help it. It will be necessary to build an adaptive framework that re-compiles methods with high abort rates resulting from program behavior changes. We discuss this as part of our future work (Section 7). Finally, in Section 8 and 9, we discuss related work and conclude, respectively.

2. MOTIVATION

Even high quality integer and enterprise code has significant inefficiencies resulting from two sources: good software engineering practice and the safety mechanisms provided by modern languages. Good software engineering practices and an emphasis on programmer productivity demand that source code be readable, debugable, maintainable, and reusable, which often translates to frequent control flow and many invocations of small virtual methods. Modern language safety features include performing NULL checks on dereferenced pointers, array bounds checks to catch array overruns, and checked dynamic casts to ensure type safety. While these checks rarely fail, their frequency significantly impacts the average basic block size, as observed by the compiler.

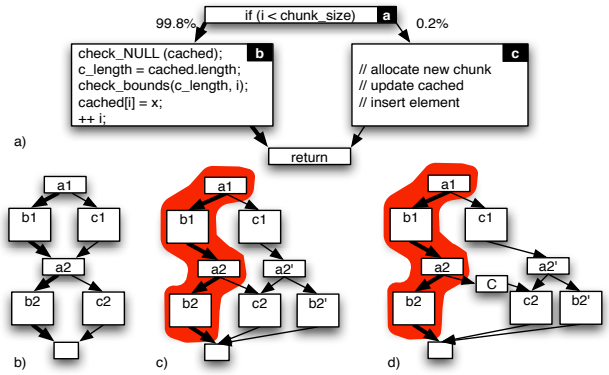

Figure 2: An example Java method with hot and cold paths. a) The hot path simply checks that the index is within the current `cached` array segment, writes an array element, and increments an index. b) The control-flow graph when two method calls are inlined. c) Superblock formation removes incoming edges from the hot path through code replication. d) Optimizations on the hot path can require the insertion of compensation code blocks on exits from the hot path.

In principle, compilers can be quite effective at mitigating these inefficiencies. Much of the inefficiency results from having to recompute values and perform checks that are redundant or are subsumed by other checks. Redundancy elimination techniques like value numbering and partial redundancy elimination eliminate these inefficiencies when they are within an optimization scope. However, because of the frequency of branches, only a fraction of the redundancy is within a single basic block, necessitating *global* (*i.e.*, inter-block) optimizations. As these optimizations must be correct over all paths, variable definitions and uses on cold paths can prevent redundancy elimination from occurring on hot paths.

Conventional speculative optimizations attempt to mitigate the constraints that cold paths place on optimizing the hot paths. As a brief example of the power of these optimizations, we describe a representative optimization opportunity in Java, taken from the DaCapo benchmark Xalan*. Figure 2(a) shows a simplified control-flow graph for the `addElement` method that inserts an integer into a `SuballocatedIntVector` object, which provides an efficient implementation of an extensible vector of integers. To avoid having to reallocate and copy the whole vector whenever the vector extends beyond its current allocation, the object maintains an array of integer sub-arrays so that the vector can be extended simply by allocating a new integer sub-array.

As is common in much of the code we analyzed, the function `addElement` has a fast hot path and a slower cold path. The fast path is invoked whenever an element is inserted into the same sub-array as was previously accessed (the software caches the most recent sub-array); since insertions are generally to sequential elements and the sub-arrays are large, this fast path ends up handling 99.8% of the calls. The slow path handles the rare cases when an access is performed to a segment other than the cached one, including when new segments are allocated. At the hottest call site, the function is called twice sequentially on the same object, as shown below:

```
m_data.addElement(m_textPendingStart);
m_data.addElement(length);
```

Inlining this method at both call sites (as shown in Figure 2(b)) can expose some redundancy to the compiler. Figure 3(a) shows the code for the hot path **a1**→**b1**→**a2**→**b2**. By performing superblock

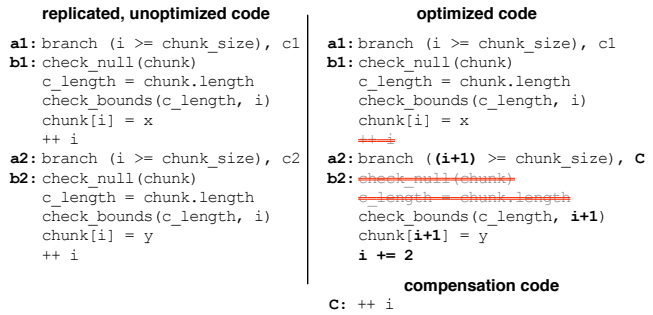

Figure 3: Compiler-based redundancy removal. (a) unoptimized code after inlining, (b) through superblock formation, the second copy of blocks **a** and **b** can be optimized knowing that the first copies will already have been executed, enabling constant folding of the first increment of `i` and removal of the redundant NULL check and load of the vector's `length` field, (c) the constant folding of the increment to `i` effectively involves downward code motion of `++ i` past the branch in block **a2**, requiring compensation code to be inserted in block **C**.

formation [10], which involves code replication, the compiler can remove the incoming edge **c1**→**a2** (shown in Figure 2(c)), so that it can guarantee that execution of block **b2** only occurs if block **b1** was executed (*i.e.* **b1** dominates **b2**). This restructuring enables the compiler to trivially remove those operations from **b2** that are redundant with those in **b1** (shown in Figure 3(b)). One of the optimizations applied (constant propagation of the first `++i`) effectively removes an instruction from the path **a1**→**b1**→**a2**→**c2**; to correct for this, however, the compiler must insert a *compensation* block, **C**, into the control flow graph as shown in Figure 2(d). The block **C** holds the removed code as shown in Figure 3(c).

While these optimizations can be relatively effective, their implementation introduces a certain amount of compiler complexity. In contrast, with a hardware atomicity primitive, the same hot path code can be generated without needing any compensation code. Furthermore, the example shown is a rather simple one; as the scope of the optimization grows, the number of hot path exits will grow (*e.g.*, Figure 1(c)) as will the number of optimizations requiring compensation at a given hot path exit. By obviating the need for compensation code, a hardware atomicity primitive eliminates the complexity resulting from corner-case interactions between compensation from two different optimizations.

3. PROVIDING HARDWARE ATOMICITY

Hardware checkpointing has been previously proposed for implementing resource-efficient high-performance processors. We survey this research and draw parallels between this microarchitectural technique and the use of hardware atomicity to optimize software execution. We then present a proposed interface for atomicity primitives and discuss their implementation requirements.

3.1 Checkpoints and Hardware Atomicity

Modern processors employ speculative execution and typically record information at a fine granularity for when speculation fails and execution state needs to be restored. However, speculation mostly succeeds, and the recorded information is not frequently needed. Checkpoint processors use this observation to optimize recovery information management [1, 4, 14]. They record recovery state at coarse intervals (100s of instructions) instead of at every instruction. When a misspeculation does occur, the processor restores the checkpoint and restarts execution, adaptively tracking information at a finer granularity after a misspeculation. This checkpoint abstraction obviates much of the fine-grain bookkeeping, since execution can always restore to a safe point.

In this work, we extend the checkpoint abstraction to incorporate hardware atomicity, ensuring that memory updates also appear to occur atomically. Hardware provides atomicity for a sequence of instructions by ensuring that either all the instructions appear to be committed at the same time or that none are. Specifically, we consider a semantic where the memory operations performed by an atomic region appear to occur instantaneously, with all other memory operations in the system appearing to occur either before or after.

Checkpoint processors do not automatically provide hardware atomicity. Previous proposals generally provide an execution that satisfies the underlying memory model, the requirements of which may be weaker than atomicity. Providing hardware atomicity for a sequence of instructions involves the following steps [18]: 1) creating a register checkpoint at the recovery point, 2) tracking all memory addresses accessed by the instructions, 3) buffering all updates performed by the instructions, 4) using an ownership-based cache coherence protocol to detect conflicting accesses from other agents, 5) discarding updates on a conflict, and 6) committing the updates in the cache atomically.

3.2 Exposing Hardware Atomicity to Software

Similar to how checkpoint processors use the checkpoint abstraction to optimize execution, we propose using the hardware atomicity abstraction to optimize execution of software. Hardware atomicity provides an atomic region with the following invariant: either the region commits successfully, or all changes performed in the region are undone and control is transferred to an alternate region. The compiler attempts to execute a speculative version of the code as an atomic region, and, if the execution is unsuccessful, it falls back to a less aggressively optimized version that includes all of the paths. Hardware limitations such as limited buffering are treated as implicit exits to cold paths, thus reducing requirements from the hardware.

The hardware atomicity also alleviates the complexity of proving the correctness of software optimizations in the presence of multiprocessing. Memory ordering is achieved by virtue of hardware atomicity.

The runtime uses three instruction set extensions to expose the atomicity invariant to the runtime.

- `aregion_begin <alternate PC>`. This instruction signals the start of a speculatively-optimized region and asks the hardware to create a recovery point (similar to a branch instruction creating a rename table checkpoint) and specifies the alternate code path for aborts.
- `aregion_end`. This instruction ends the region and commits updates atomically.
- `aregion_abort`. This instruction permits the runtime software to explicitly rollback changes during the execution to a prior point. This is used when an assertion fails and the execution must proceed down a cold path.

Figure 4 illustrates the use of these new instructions. Causes for atomic region aborts are communicated to the software via two additional registers. The first register encodes the reasons for an abort (*e.g.*, explicit abort, interrupt, data conflict, exception, etc.). The second register records the program counter of the instruction responsible for an abort (if any). This information allows the JVM to diagnose the cause of aborts and adaptively recompile, in order

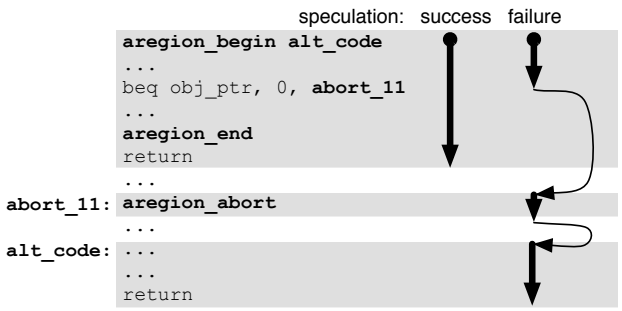

Figure 4: Example usage of the atomic primitives by the generated code If a speculation succeeds, no abort conditions will be invoked and the execution will reach an `aregion_end` that commits the atomic region. Speculation fails when an abort condition evaluates to true, causing a branch to be taken to an unconditional abort instruction. When the abort instruction commits, the register state is restored, the lines written in the atomic region are invalidated, and control is transferred to an address specified by the `aregion_begin` instruction.

to maintain a low misspeculation rate. Such a diagnosis requires the compiler to generate a unique abort instruction for each assertion and maintain a mapping from the program counter of the abort instruction to the corresponding assertion in the compiler's intermediate representation.

3.3 Microarchitectural Implications

A critical aspect of the hardware atomicity abstraction is its synergy with high-performance processor implementations. This is important since any proposal for exposing hardware mechanisms to software must also be amenable to high-performance implementations. By using the checkpoint substrate, we can build high-performance processors that can execute atomic regions at a high rate.

A simple abstraction also provides significant flexibility to hardware designers. The atomic abstraction allows hardware to execute the code region in whatever way seems fit, as long as when an abort condition occurs, the execution restores to the beginning of the region with appropriate information in the appropriate registers. Since the common and fast path execution is synergistic with a checkpoint processor, overhead during fast path execution can be reduced to that of a checkpoint processor with atomicity support. This hides the latencies introduced due to any ordering-based serializations, the same way checkpoint processors do so. Other work has shown that serializing operations and instructions that disrupt the smooth flow of instruction execution through the pipeline degrade performance [3] by introducing delays in the pipeline.

Various implementation strategies based on checkpoint architectures exist for hardware atomicity. In our implementation, we assume the data cache retains the data footprint of the atomic region and a register rename table checkpoint is used for recovering register state. Each cache line is extended with two bits for tracking which addresses have been read and written in the atomic region. These addresses are exposed to the coherency mechanism to observe invalidations. Flash clear operations are used to commit and/or abort speculative state.

While support for hardware atomicity may appear similar to hardware support for transactional memory [12], significant differences in requirements and usage exist, resulting in different hardware implementation requirements. Transactional memory is proposed primarily for scalability and can potentially tolerate some loss of single-thread performance to achieve this scalability. In contrast, our use of hardware atomicity is focused on high single-thread performance, and any execution overhead of the atomic region will reduce the benefit of these optimizations. We discuss the implications of simplified implementations on our usage model in Section 6.3. Use of a checkpoint execution substrate for implementing hardware atomicity allows us to achieve a nearly no-overhead common case execution and permit multiple atomic regions to be inflight simultaneously.

Apart from the performance goal of fast common case execution, our usage model simplifies the functionality required from the hardware implementation. Since the hardware is being opportunistically used to improve the performance of a single thread, a best effort implementation is sufficient.

4. FORMING AND OPTIMIZING REGIONS

This section focuses on how the compiler uses the atomic region abstraction to generate better code. Specifically, we demonstrate how support for atomic regions can be introduced into a compiler without significant changes, an algorithm for selecting appropriate atomic regions while achieving good program coverage, why assertions constrain optimization significantly less than branches, and optimizations enabled by atomic regions.

Atomic regions and abort as try/catch: Modern languages like Java generally provide support for structured exception handling, which in Java takes the form of `try` and `catch` blocks. These primitives enable the programmer to specify one block of code that should be executed assuming that no exceptions occur and another one to be executed to handle an exception. To support these language features, a compiler must be able to represent them in its intermediate representation (IR).

One of the most important observations that we made in this work is that support for `try` and `catch` directly corresponds to what is required to represent both atomic regions and the abort path to non-speculative recovery code. This observation reduces the problem of supporting software speculation within the compiler to that of simply transforming the program's control flow graph so that atomic regions look like try blocks and non-speculative recovery code looks like a catch block. As a result, *no optimizations needed to be modified to start exploiting the optimization opportunity exposed by the atomic regions*. Our entire implementation (including the implemented transformations and optimizations) required approximately 3,000 lines of code (LOC) ($\sim$3% of the optimizing compiler), roughly two-thirds of which is the region selection algorithm. While the complexity of atomic region formation corresponds closely that of reported by Hwu et al. for superblock formation (2,000 LOC), their superblock optimizations incurred an additional 12,000 LOC [10].

Region formation: In selecting regions for optimization, our implementation maintains three properties: 1) overly large regions must be avoided, 2) atomic regions must not be nested, and 3) atomic regions will be single-entry, multiple-exit subgraphs, containing arbitrary intraprocedural control flow. The first property permits a best-effort implementation of atomicity (*i.e.*, atomic regions that overflow the cache or receive an interrupt will abort) as well as bounds the lost effort when a region aborts. We avoid nesting, in part, to demonstrate that its support is not a hardware requirement. In addition, nesting only occurs as a result of encapsulating a non-inlined call within an atomic region and we have yet to observe a case where this will significantly improve optimization. The last property simplifies region formation by building upon other well understood single-entry techniques but without the control flow limitations imposed by building regions from traces [7, 10, 16].

The process of region formation is fundamentally a profile-driven one. Our goal is to select regions for optimization that exclude infrequently executed (or "cold") code paths. As is typically done in JVMs, the first-pass compiler inserts instrumentation to profile program behaviors (*e.g.*, branches, virtual calls). For our experiments, we define as *cold* any paths whose branch bias is less than 1%; these paths will be removed from atomic regions. We use the term *non-cold* to refer to all paths that are not cold.

Our region formation process has five steps:

Step 1. Aggressively inline methods
Step 2. Select region boundaries (See Algorithm 1)
Step 3. Replicate flowgraphs for selected regions
Step 4. Convert cold edges into asserts
Step 5. Remove all inlined methods from non-speculative paths

The first step enlarges the optimization scope by aggressively inlining methods. We do this without fear of the "code bloat" typically associated with inlining for two reasons. First we will only retain an inlined method along a speculative path if it is contained entirely within an atomic region (we proactively prune inlined methods that do not satisfy this criteria). Second, the remaining inlined methods will have their infrequently executed paths speculatively removed, enabling the retained paths to be further reduced in size by optimization.

The next step, selection of region boundaries is the crux of region formation and is specified in detail in Algorithm 1; here we overview its operation. The goal of boundary selection is to identify a set of blocks that will become the entry and exit points for atomic regions. More specifically, it focuses on identifying blocks that should become atomic region entries. The placement of atomic region exits is largely born of necessity (*i.e.*, atomic regions are terminated at precisely the points that they could not be extended beyond without violating one of our invariants).

Placement of atomic region boundaries starts by considering loops to decide whether individual loop iterations should be executed in atomic regions or whether the whole loop should be encapsulated within a single atomic region. There are two factors which influence the decision: the dynamic path length through the loop and whether the loop contains a non-inlined call on a non-cold path (*i.e.*, will not be speculatively removed). We choose per-iteration atomic regions when loop iterations are large or if the average number of iterations executed is high enough that the region might overflow the cache. As we terminate atomic regions at non-inlined calls and often begin new ones immediately after the call returns, if such a call is on a non-cold path within the loop, we must insert an atomic region boundary in the loop's header to prevent the creation of irreducible flowgraphs.

Next, the region selection *un-inlines* any of the methods that were aggressively inlined in Step 1 that will not be completely encapsulated in atomic regions; this step prevents "code explosion" resulting from the method needing to be fully included on an atomic region's non-speculative path. If one of these methods includes an atomic region boundary (from the previous step) or a non-inlined call on a non-cold path, it is un-inlined.

The last part of the boundary selection algorithm places boundaries along acyclic paths. The algorithm iteratively selects the hottest block that has not already been visited and traces the dominant path through the block, terminating the trace at already selected region boundaries or at the method entry, exit or call continuations. All loop pre-headers and loop exits contained on the dominant path, as well as the start and end of the path, are candidates for boundary selection. The algorithm selects the subset of the candidate boundaries that minimizes Π in Equation 1 where R is the desired region size and r_n is the size of the n^{th} candidate region (this equation was originally used in the task selection algorithm for MSSP [23]).

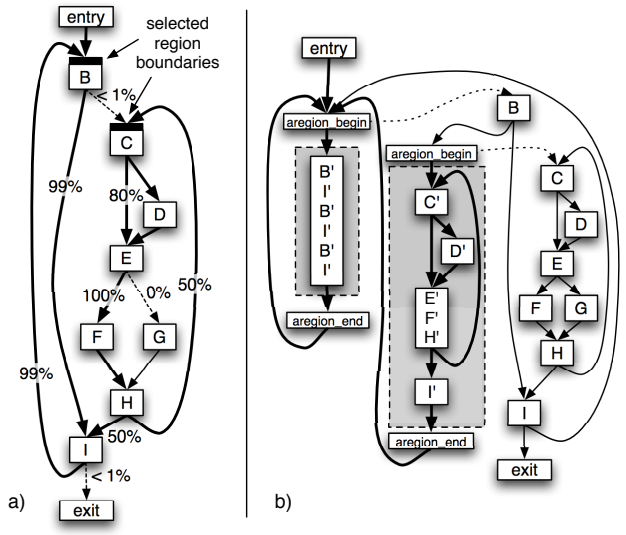

Figure 5: Region formation: (a) before, (b) after.

$$\Pi = \sum_{n=1}^{N} \frac{(R - r_n)^2}{R \cdot r_n} \quad (1)$$

Once atomic region boundaries have been selected, Step 3 creates the atomic regions by performing a depth first search (ignoring cold paths) starting from each selected region boundary, stopping at other selected region boundaries, the method exit, and any non-inlined calls and then copying the visited blocks. An `aregion_begin` is placed at the entry to the region, and an `aregion_end` is placed at each region exit. All edges into the block that the region entry was copied from are moved to the `aregion_begin` and an exception edge is added from the `atomic_begin` to the source block. Figure 5(b) shows the result of this step (partial loop unrolling has been applied to the outer loop).

The remaining steps of atomic region formation convert cold branches into asserts (Step 4) and replace inlined methods on non-speculative paths with calls (Step 5).

As we initially stated, we avoid generating large atomic regions and we have found that setting LOOPPATHTHRESHOLD and R to a value of 200 high-level intermediate representation operations[1] satisfies this property without sacrificing much opportunity.

Why asserts constrain optimization less than branches do: As the final step of our region formation, we convert branches from the hot path to the cold paths into assertions in the compiler's intermediate representation (IR). Previously, we claimed that these assertions constrain optimizations less than branches; we now explain why this is true. In our high-level IR, the assertion operations are implemented as simple operations that have only source operands and no side effects, like an ALU operation that produces no value. They, unlike branches, can be completely ignored when optimizing other instructions. Furthermore, they can be optimized by existing passes: they can be freely scheduled (limited only by their data dependences and the boundaries of the atomic region) and redundant asserts are eliminated by existing redundancy elimination passes such as global value numbering. Only dead code elimination needs to be informed that these operations are *essential* and should not be removed.

[1]This has a loose correspondence to the number of hardware instructions actually generated

Algorithm 1 Selection of atomic region boundaries

```
procedure SELECTBOUNDARIES(method)
    selectedBoundaries ← ∅                                              ▷ Set of blocks

    // Place region boundaries at the headers of large loops (i.e. those with long iterations
    // or high trip counts) and loops containing calls reachable along non-cold paths
    L ← LOOPSINPOSTORDER(method)                                        ▷ Process loops from innermost to outermost
    foreach loop in L do
        loopBlocks ← GETBLOCKS(loop)
        loopHeader ← GETLOOPHEADERBLOCK(loop)
        hasWarmCall ← HASCALLONWARMPATH(loopHeader, loopBlocks)         ▷ Is a call reachable along non-cold paths?
        loopPreHeader ← GETLOOPPREHEADERBLOCK(loop)
        loopPathLength ← LOOPWEIGHT(loop) / GETEXECCOUNT(loopPreHeader) ▷ LOOPWEIGHT defined in Algorithm 2
        if (loopPathLength ≥ LOOPPATHTHRESHOLD) or hasWarmCall then
            selectedBoundaries ← selectedBoundaries ∪ {loopHeader}

    // Prune inlined methods that contain selected loops or calls reachable along non-cold paths.
    // This limits unnecessary code bloat and is part of our partial inlining implementation
    foreach inlinedMethod in INLINEDMETHODS(method) do
        inlinedBlocks ← GETBLOCKS(inlinedMethod)
        inlinedEntry ← GETENTRYBLOCK(inlinedMethod)
        inlinedContinuation ← GETCONTINUATIONBLOCK(inlinedMethod)
        hasWarmCall ← HASCALLONWARMPATH(inlinedEntry, inlinedBlocks)    ▷ Is a call reachable along non-cold paths?
        hasSelectedLoop ← (selectedBoundaries ∩ inlinedBlocks) ≠ ∅
        if hasWarmCall or hasSelectedLoop then
            UNINLINEMETHOD(inlinedMethod)

    // Place region boundaries along acyclic paths
    visited ← ∅
    traceBoundaries ← { GETENTRYBLOCK(method), GETEXITBLOCK(method), GETCALLBLOCKS(method) }
    maxBlockExecCount ← GETMAXBLOCKEXECCOUNT(method)
    B ← BLOCKSSORTEDBYEXECCOUNT(method)                                 ▷ Process most frequently executed blocks first
    foreach block in B do
        if block ∉ visited and GETEXECCOUNT(block) ≥ (maxBlockExecCount/100) then
            dominantPath ← TRACEDOMINANTPATH(block, selectedBoundaries ∪ traceBoundaries)   ▷ Defined in Algorithm 2
            acyclicBoundaries ← SELECTACYCLICBOUNDARIES(dominantPath)   ▷ Selects boundaries that minimize Equation 1
            selectedBoundaries ← selectedBoundaries ∪ acyclicBoundaries
            visited ← visited ∪ dominantPath

    return selectedBoundaries
```

Atomic regions enable optimizations: The guarantees provided by atomic regions enabled us to implement several additional optimizations: partial inlining, partial loop unrolling and speculative lock elision[2] [18]. The implementations of partial inlining and partial loop unrolling were enabled by the design simplicity offered by atomic execution, and speculative lock elision was enabled by the atomicity and isolation guarantees provided by hardware. The relatively small amount of code required to implement these optimizations (∼200 LOC each for partial inlining and partial loop unrolling, and ∼400 LOC for speculative lock elision) demonstrates the simplicity offered by atomic regions.

Partial inlining (loop unrolling) exposes additional opportunity by enlarging the optimization scope, but limits static code expansion by obviating the need to inline (unroll) infrequently executed paths in the method (loop). However, implementing either optimization without atomic regions overly burdens the compiler writer with the responsibility of guaranteeing that the correct program state can be recovered and forward progress made if an infrequent path is executed. With atomic region support, the implementation of both partial inlining and loop unrolling becomes almost trivial. The hot paths of inlined methods and loops are simply wrapped in atomic regions and the infrequent paths are converted into assertions. If an infrequent path is executed, an assert will fire, hardware will redirect execution to the corresponding non-speculative code (that has not been inlined or unrolled).

[2]We use SLE to reduce monitor overhead, but our optimization would also reduce monitor-induced serialization in multithreaded workloads

Speculative lock elision (SLE) exploits opportunity exposed by our atomic region formation. Atomic regions often contain balanced pairs of Java monitor enter and exit operations, and these monitors are typically uncontended. The JVM we used already provides fast-path implementations for common lock behaviors using reservation locks [11], but even the fastest path must still check the status of the lock and update it with a store (both at monitor entry and monitor exit) to track lock nesting depth. We can reduce this monitor overhead with atomic regions; when a balanced pair of monitor operations is contained within an atomic region, our implementation of SLE must only load the value of the lock upon monitor entry and verify—a compare and branch—that it is not held by another thread. In the common case, no action is needed at the monitor exit. This improvement to single-thread performance is in addition to any concurrency benefits from optimistically executing a synchronized method/block.

5. EXPERIMENTAL METHOD

Evaluating the performance impact of run-time compiler enhancements using new hardware features presents a number of challenges. First, in the absence of real hardware, a full-system simulator is a necessity, as a JVM and some of the workloads are multi-threaded and use many system features. Second, because the compilation is performed *during* the program run, the benchmark runs have to be sufficiently long for the staged optimizer to produce the fully optimized code. Finally, because we are comparing the performance of two different compilers, we need to select equivalent regions of the program's execution to make an "apples-to-apples" comparison.

Algorithm 2 Used during selection of atomic region boundaries

```
// Generate an ordered list containing the most frequently executed
// path through the specified block. Stop tracing at selected
// boundaries and trace boundaries
procedure TRACEDOMINANTPATH(seedBlock, traceBoundaries)
    dominantPath ← [seedBlock]
    traceBlock ← seedBlock; done ← false
    while ¬done do
        traceBlock ← GETDOMINANTOUTEDGE(traceBlock)
        dominantPath ← dominantPath + [traceBlock]
        if traceBlock ∈ traceBoundaries then
            done ← true
    traceBlock ← seedBlock; done ← false
    while ¬done do
        traceBlock ← GETDOMINANTINEDGE(traceBlock)
        dominantPath ← [traceBlock] + dominantPath
        if traceBlock ∈ traceBoundaries then
            done ← true
    return dominantPath

procedure LOOPWEIGHT(loop)
    weight ← 0
    foreach block in GETBLOCKS(loop) do
        blockExecCount ← GETEXECCOUNT(block)
        numBlockOps ← GETNUMOPERATIONS(block)
        weight ← weight + (blockExecCount * numBlockOps)
    return weight
```

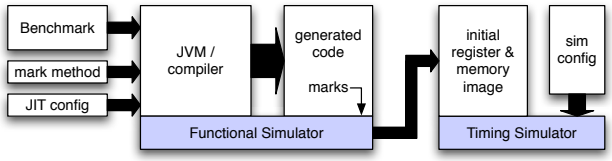

Figure 6: A infrastructure for performance analysis of JVMs on unimplemented hardware platforms.

Figure 6 shows our evaluation infrastructure, which performs the evaluation in two steps. First, the benchmark is executed on our modified version of the Apache Harmony*DRLVM Java Virtual Machine (JVM) [9] on the SoftSDV*full-system simulator, which we extended to support our ISA extensions for atomic regions as discussed in Section 3[3]. We use the DRLVM's `server` execution manager configuration to maximize code quality; the whole process is completely automatic and profile driven. This functional simulation is run for a sufficiently long duration to allow any compilation to be performed during the run and for the staged optimizer to produce the fully optimized code.

Once we get to a representative portion of the execution, we record the state of the functional simulation for use in timing simulation. The format of the state recorded, known as LIT, contains a snapshot of the initial processor architectural state and memory as well as a trace of all system interrupts necessary to simulate system events such as DMA traffic etc. The LIT is consumed by a detailed execution-driven simulator working on top of a micro-operation (uop) level IA-32 architecture simulator. This simulator performs a timing simulation, including accurate modeling of a detailed memory subsystem, wrong path execution, interrupts, system interactions, and DMA events. Our baseline **4-wide OOO** processor parameters are shown in Table 1. A checkpoint execution substrate, similar to that of checkpoint processors, provides atomic execution.

[3]For debugging the compiler, we also developed a means to test on real machines by registering a signal handler for invalid opcode exceptions (triggered by the unrecognized `aregion_begin` instructions) that inspects the faulting instruction and branches immediately to the (non-speculative) recovery path.

Processor frequency	4.0 GHz
Rename/issue/retire width	4/4/4
Branch mispred. penalty	20 cycles
Instruction window size	128
Scheduling window size	64
Load/store buffer sizes	60/40
Functional units	Pentium® 4 equivalent,
Branch predictor	combine: 64K gshare/16K bimod
Hardware data prefetcher	Stream-based (16 streams)
Trace Cache	64 K-uops, 8-way
I-TLB	128 entries
D-TLB	64 entries, 4-way
L1 Data Cache	32 KB, 4-way, 4 cycle hit, 64B line
L2 Unified Cache	4 MB, 8-way, 20 cycle hit, 64B line
L1/L2 Line size	64-bytes
Memory latency	100 ns

Table 1: Baseline processor parameters

Benchmark	Description	#
antlr	Generates parser/lexical analyzer	4
bloat	Bytecode analysis and optimization tool	4
fop	Parses/formats XSL*-FO to generate PDF*	2
hsqldb	Executes JDBCbench*-like benchmark	1
jython	Interprets pybench Python benchmark	1
pmd	Analyzes a set of Java classes	4
xalan	Converts XML* documents into HTML*	1

Table 2: DaCapo benchmarks used in evaluation. # = number of samples used in evaluation.

Since we are comparing the performance of two different compilation approaches, we must select equivalent regions of the program's execution. We modify the compiler to insert special markers that can be interpreted by the full-system simulator. These markers bound equal work at the program level, thus allowing a fair comparison. To select good marker locations, we collect a complete trace of method invocations from the benchmark's execution. We break this trace into groups of 10,000 methods and use SimPoint 3.0's phase classification tool [8] to identify phases. For up to four phases per benchmark, we select a *marker method* that can be used to bound a simulation sample and that is infrequently invoked (so that it minimally perturbs the execution). Three dynamic invocations of the marker method are used to identify the sample: i) the beginning of the warm-up period, ii) the end of warm-up/the beginning of timing simulation, and iii) the end of timing simulation. This method has some similarities to concurrent work [17].

When the JVM is invoked, we pass the marker method identifier—the class name, method name, and call signature—to the JVM, which compares it to each method compiled and inserts the marker in the appropriate method's prologue. While the exact number varies, warm-up and simulation intervals are selected to contain on the order of millions to tens of millions of instructions. For the benchmarks with multiple important phases, we report data by weighting the results for each sample by its phase's contribution to the overall execution.

We use the DaCapo benchmark suite [2] for evaluation (version dacapo-2006-10). The suite is intended for evaluation of JVMs by the programming languages, memory management, and computer architecture communities and consists of a set of open source, real world applications with non-trivial memory loads. Table 2 lists the benchmarks used and their descriptions. The remaining benchmarks were not included for experimental method reasons: `chart` and `eclipse` were too long running, `luindex`'s samples could not be validated in time, and `leusearch` is non-deterministic. To

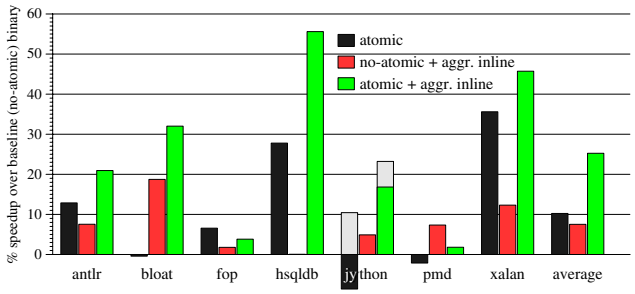

Figure 7: Execution time speedups. All runs use the same hardware configuration, performance differences result from increased optimization effectiveness.

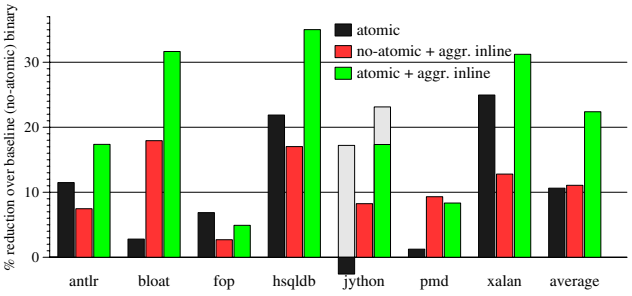

Figure 8: Micro-operation (uop) reduction.

avoid non-determinism in `xalan` we used the single-threaded version from the beta-2006-08 release of the benchmarks. To work around a bug in Harmony, we also used `jython` from the same beta release.

6. RESULTS

In our analysis, we are concerned with two metrics: performance and dynamic micro-operation (uop) count reduction. Our performance measurements compare execution time of the sampled regions. We measure the reduction in dynamic uop counts, because they, in general, will directly translate into energy efficiency. Fewer uops flowing down the pipeline will result in less switching activity, which in turn results in a reduction in the amount of energy consumed to perform a given unit of program work.

Initially, we compare two compiler configurations: `no-atomic`, a baseline set of optimizations that corresponds closely to Harmony's default server configuration, and `atomic`, which is our optimizer enhanced to exploit hardware-supported atomic execution. The optimization passes enabled are the same for both, except `atomic` performs atomic region formation, partial inlining, partial loop unrolling, and speculative lock elision.

As shown in Figure 7, we find that these optimizations enable a significant (10% average) speedup across our benchmarks. Furthermore, these speedups are accompanied by a nearly comparable reduction in the number of uops retired, as shown in Figure 8. By providing a simple recovery abstraction (atomic regions) the hardware has facilitated the compiler's generation of higher performance and more efficient code.

When we inspected the code, we clearly saw evidence of speculative optimizations, even though the compiler had no such optimizations implemented in their traditional formulations. In one atomic region, for example, elimination of cold paths enabled the compiler to simplify an indirect branch to a conditional branch (as only 2 of the 9 cases were not-cold), eliminate branches via constant propagation previously inhibited by cold control flow, eliminate redundant loads, and eliminate redundant checks. Some of the benefits in other regions, however, were merely the result of increasing the scope of optimization through (partial) inlining and (partial) loop unrolling beyond what the baseline inliner and loop unroller were exposing. In order to demonstrate that this scope enlargement is not responsible for all of our benefit, we ran two additional sets of experiments: the baseline and `atomic` configurations with an unrealistically large inlining threshold (a factor of five larger than the baseline), which should achieve the optimization potential resulting just from scope enlargement. The results for these configurations are also shown in Figures 7 and 8.

Bench.	Atomic Regions			Region Abort Rate	
	coverage	unique	size	%	per 1k uop
antlr	9%	96	47	0.02	0.0004
bloat	69%	93	128	4.3	0.12
fop	20%	73	32	0.01	0.0007
hsqldb	76%	75	88	2.74	0.24
jython	87%	14	227	0.69	0.27
pmd	32%	32	42	2.2	0.18
xalan	78%	37	78	0.28	0.03

Table 3: Atomic region statistics. coverage: fraction of executed uops in atomic regions, **unique:** average number of unique atomic regions in execution sample(s), `size`: average size of atomic regions (in dynamic instructions), **abort %:** percentage of regions aborting, **aborts/1k uop:** number of aborts per 1,000 uops. Data shown for the `atomic+aggressive inlining` configuration.

From this data it can be clearly seen that the atomic region-based optimizations are achieving more than just scope enlargement, as the performance from `no-atomic+aggressive inlining` is less than half of the `atomic+aggressive inlining` case. Actually, increasing the optimization scope appears to disproportionately benefit the `atomic+aggressive inlining` case, as its speedup (25.3%) is more than the sum of those of `atomic` and `no-atomic+aggressive inlining` (10.2% and 7.5%, respectively).

6.1 Understanding the Variation

Clearly, our optimizations do not uniformly benefit all of the benchmarks; the speedups achieved by the `atomic+aggressive inlining` configuration range from 56% (`hsqldb`) to 2% (`pmd`). In this section, we explore the sources of this variation.

Across the benchmarks, we see a strong correlation between the uop reduction and speedup, which is not surprising as both generally occur when code is optimized more effectively. It should be noted that our optimizations are not just removing a subset of the instructions (as is done in SlipStream [20]); in many cases we are also replacing uops with other, simpler uops (*e.g.*, SLE replaces compare-and-swap primitives and monitor data structure updates with a load and a branch) as well as simplifying the code's critical path. This explains why many of the benchmarks exhibit superlinear speedups relative to their uop reduction.

The biggest factor affecting the degree of optimization seems to be coverage. Table 3 shows that four of the benchmarks with the high speedups—`bloat`, `hsqldb`, `jython`, and `xalan`—execute most (upwards of 69%) of their uops in atomic regions. As we are reporting coverage **after** optimization and most of the reduction in dynamic uop count occurs in the atomic regions, an even larger fraction of the program is actually encapsulated in atomic regions than these coverage numbers suggest; this effect also explains how `antlr` can achieve a 17% uop reduction with only 9% coverage. These four benchmarks also have the largest atomic regions. With average region sizes ranging from 75 to 225 instructions *after optimization* there is sufficient scope for significant optimization.

The outlier from this trend is `antlr`, which manages to achieve significant speedups despite low coverage because a large fraction of the instructions are eliminated from the atomic regions it does form. On average, two-thirds of the instructions in `antlr`'s atomic regions get optimized away. `antlr`, like the other benchmarks that get significant speedups, is getting most of its benefits from two main sources: generic redundancy elimination and elimination of monitor overhead of calls to synchronized `classlib` methods.

The `pmd` benchmark actually slows down in the `atomic` configuration, because it has relatively low coverage, but incurs a 2.2% abort rate for its atomic regions. This relatively high abort rate is the result of a behavioral change in four atomic regions that occurs between when the behavior is profiled and where our execution sample is taken. Such a change incurs aborts when a path that initially appears cold is removed from the atomic regions and then later starts to be frequently executed. Such behavior changes have previously been documented [19] and their negative impacts on performance can be eliminated through adaptive recompilation when an atomic region begins to frequently abort [22].

Two other benchmarks—`hsqldb` and `bloat`—also have non-trivial abort rates, but achieve significant speedups despite them. In `hsqldb`, the aborts occur very early in the atomic region so they have little negative impact beyond a pipeline flush. In `bloat`, they do have a large impact; almost all of `bloat`'s aborts occur in one of its four execution samples—the one from the least dominant phase—and that sample incurs a 33% slowdown. Without that phase, `bloat`'s speedup would be 40% (up from 32%) for the `atomic+aggressive inlining` configuration.

Despite performing well in the `atomic+aggressive inlining` configuration, the `jython` benchmark incurs a slowdown in the `atomic` configuration. The source of this discrepancy, is an important method (`getitem`, called four times in a hot loop) that is not being inlined by the partial inliner in the `atomic` configuration. This results in a large number of small atomic regions being formed that incur more overhead than they provide optimization opportunity. The `getitem` method is not being inlined because it contains what appears to be a polymorphic call site and our algorithm will not partially inline methods containing polymorphic calls. If `getitem` were inlined, however, this call site is perfectly monomorphic. If we force our implementation to recognize this fact, `getitem` is inlined and the `atomic` configuration's 9% slowdown becomes a 10% speedup, as shown by the grey bars in Figure 7. These performance benefits may also be achieved through implementing an adaptive recompilation strategy, as we can perform aggressive speculation (*e.g.*, that polymorphic call sites are monomorphic) and recompile those methods containing atomic regions that frequently abort.

6.2 Architectural Analysis of Atomic Regions

In this section, we quantify the atomic regions generated by the compiler from the hardware's perspective. In terms of implementing atomicity in hardware, it is important to know the size of atomic regions in terms of dynamic instruction count and data footprint.

If the compiler-generated atomic regions were consistently small, they could be buffered completely within the pipeline, but we find this not to be the case. A 128-entry reorder buffer is insufficient to hold nearly 25% of the atomic regions executed (data not shown). A small fraction of atomic regions even contain over 1,000 uops. By using register checkpoints for recovery (similar to branch checkpoints but those that live past speculative retirement), we enable the compiler to construct such regions.

Our implementation uses the data cache to buffer the reads and writes in the atomic regions, similar to prior work [5, 14]. We find that the modern L1 caches are easily sufficient for holding the read and write set of the atomic region. Most atomic regions access less than 10 cache blocks and 50 cache blocks is sufficient for 99% of the atomic blocks we observed (for reference a 32KB cache with 64B blocks holds 512 blocks). Only 110 atomic regions out of the 1.7 million that we simulated touched more than 100 cache blocks and only one overflowed the cache. Clearly, our region selection algorithm is effective at tolerating the constraints of a bounded atomic primitive. While the read and write sets of the atomic regions fit easily within the cache, the number of loads and stores, which tend to be proportional to the number uops in the atomic region, are generally too large for fully buffering in the load and store buffers.

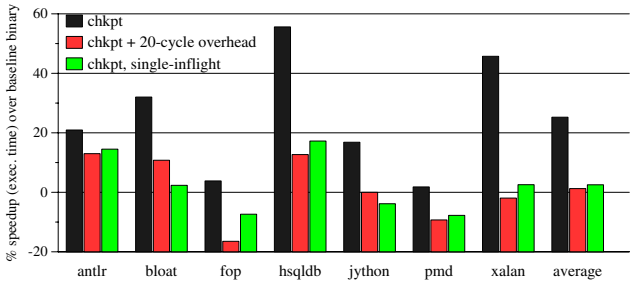

Figure 9: Sensitivity to hardware atomic primitive implementation. All runs use the same code (`atomic+aggressive inlining`) on different hardware configurations: **chkpt:** the base high-performance non-stalling checkpoint execution substrate, **+ 20-cycle:** stalls the pipeline for 20 cycles at every `aregion_begin`, and **single-inflight:** stalls an `aregion_begin` at decode if another uncommitted atomic region is already in the pipeline.

6.3 Microarchitectural sensitivity

Because this application of atomic regions is intended to improve single-thread performance, they must be implemented with minimal overhead in order to preserve the benefits achieved by the compiler optimizations. So far, we have assumed a checkpoint execution substrate (similar to checkpoint processors) to implement hardware atomicity in a high performance manner.

Now we investigate impact on performance of alternate or simplified implementations in the absence of a high-performance checkpoint substrate to provide atomicity. In particular, we are concerned about two potential sources of overhead: overhead in the form of additional operations and serialization that may occur as part of the `aregion_begin` to record recovery state, and serializing overheads that may occur due to simplified implementations of the `aregion_begin` and `aregion_end` instructions in the absence of a checkpoint substrate. We explored the performance sensitivity to these effects by modeling two ways such overheads may be exposed. First, we measured the performance of the `atomic+aggressive inlining` compiler configuration with a simulator configured to stall the pipeline for 20 cycles when processing an `aregion_begin`. Second, we considered implementations that only permit a single atomic region to be in flight at a time; an `aregion_begin` is stalled at decode until all preceding atomic regions commit. As shown in Figure 9, both of these configurations effectively eliminate the benefit of atomic regions in our benchmarks. The sole exception is `antlr`, which shows limited sensitivity because its execution uses atomic regions rather sparingly.

In addition to our baseline processor configuration, we measured performance on two more-modest microarchitectures, as might be incorporated into future "many-core" processors: a **2-wide OOO**

version of the baseline machine (widths reduced to 2/2/2) and a **2-wide half OOO** configuration that halves the superscalar width and all other processor structures (including caches and TLBs). We found that the relative speedups achieved by our atomic region-based optimizations closely tracked the **4-wide OOO** results shown in Figure 7 (generally within a percent or two), so the data is not shown due to space limitations.

6.4 Limitations of the existing compiler

In the process of implementing our atomic region-based optimizations, we found that sometimes the benefits of our optimizations were mitigated by limitations in the compiler's other optimizations and code generation. One particularly spectacular example of this effect occurred when we tried to remove the garbage collection *safe point* from loops completely encapsulated in atomic regions, replacing it with a single load of thread's local `yield` flag in the loop's pre-header. As it turns out, the JVM's register allocator implicitly relied on the call to the `yield()` function to prevent the registers within the loop from being assigned to variables only used outside the loop. If this call was removed, performance plummeted because many of the frequently accessed variables within loops were now being spilled to the stack.

As such, our performance results should not be considered a definitive characterization of the potential of atomic regions. We believe that significant further optimization potential exists. Nevertheless, it is important to recognize that atomic regions primarily facilitate the optimization phase of the compiler, and must be complimented by high quality code generation and run-time services to achieve high performance.

7. FUTURE WORK

While we find these initial results to be promising, they also demonstrate there is further research to be done: i) fine-tuning the region selection algorithm to achieve higher coverage and to form larger atomic regions with more optimization potential, ii) designing and implementing an adaptive framework to maximize speculation while minimizing abort rate, and iii) formulating existing optimizations for atomic regions and identifying new optimizations that atomic regions enable. We briefly elaborate on the last two items.

Maximizing the performance of atomic regions will require continuously monitoring their abort rate, and adaptively recompiling methods when their profiles change. While managed language environments like the JVM profile extensively and adaptively recompile methods based on their execution frequency, they generally do not re-compile fully-optimized methods if the profile changes because of the execution overhead of profiling the fully optimized code. This however will be a necessity when using atomic regions, as an abort rate of even a few percent can have a significant impact on performance. The process of tracking changing profiles, however, is a rather simple one in the context of atomic regions, as, for the most part, profiling is needed only when a region aborts and the hardware reports which assertion is failing. The remaining challenge, beyond engineering the system, is in computing an abort *rate* from a measured abort *count*. Doing so will require efficiently estimating the number of times each atomic region is executed, and, based on the data in Figure 9, straight-forward instrumentation of every atomic region may incur too much overhead.

We believe compiler writers will find new optimizations enabled by atomic regions, beyond SLE. One specific example is considering a post-dominance relationship within an atomic region as equivalent to a dominance relationship with regards to array bounds check elimination. In general, check elimination optimization can remove a check B if another check A exists that subsumes it (*i.e.*, it is equivalent or stronger) and A *dominates* B (*i.e.*, all paths to B go through A). With atomic regions, it becomes safe to remove a check A that is post-dominated by a subsuming check B (*i.e.*, all paths from A go through B), because if B fails, control will be transferred to a non-speculative version of the code that will test both A and B and report the failing check properly to the run time. This optimization would enable the removal of `check_bounds(c_length, i)` from Figure 3 because in the atomic region it will be post-dominated by the stronger check `check_bounds(c_length, i+1)`.

8. RELATED WORK

Whereas prior work on hardware checkpoint/rollback support for optimization has only been in the context of dynamic binary optimizers, our work demonstrates its integration into traditional compilation.

The rePLay framework [7,16], converted predictable control flow into assertions that were implemented as instructions that abort and rollback the current region. By doing so, the optimizer did not have to generate compensation code for the cold-path exits from an optimized region. However, rePLay used these assertions in a hardware-only optimizer. A modified branch predictor identified predictable instruction traces, called *frames*. A hardware code optimizer (similar to a compiler) processed these traces and applied optimizations. These traces were cached in a *frame cache*, and a frame predictor controlled dispatch.

Our proposal provides three key advantages over rePLay. First, rePLay required significantly more hardware. It used dedicated hardware for profiling, trace construction, filtering, optimization and caching. In contrast, we perform these in software. Second, because rePLay used hardware for optimizations, to manage hardware complexity it truncated frames at unpredictable branches, thus restricting the optimization scope. In contrast, we can form regions with arbitrary internal control flow. Third, since rePLay used hardware for frame caching, the optimizations were not persistent and programs with large instruction footprints would thrash the frame cache. These re-constructions and re-optimizations expend energy. In contrast, our proposal stores the optimized regions in program memory, where they persist throughout the program's execution.

The Transmeta Crusoe*processor also provides checkpoint and rollback capability for compiler optimizations [6] in the form of a shadow register file and a gated store buffer. Unlike our proposal, the Transmeta Code Morphing Software*(CMS) does not use the checkpoint/rollback support for explicit control speculation (*i.e.*, no explicit abort); it uses aggressive predication to form hyperblocks. Instead, the checkpoint/rollback capability is used to implement precise architectural exceptions and for recovery when loads are incorrectly scheduled ahead of stores.

Our proposal has three additional advantages over the rePLay and Transmeta/CMS works. First, because they were implemented in the context of a dynamic binary translator/optimizer, they are constrained to perform optimizations at the ISA level. In contrast, our approach permits higher-level optimizations including speculative lock elision and partial inlining. Second, because our proposed hardware support records the read set of the optimized region, tracks it for coherence conflicts, and commits the region's stores atomically, it is multi-processor safe. Finally, the size of our regions is less constrained than in rePLay (no conditional branches and up to the size of the re-order buffer) and Crusoe (limited by the size of the gated store buffer and memory disambiguation hardware); the only size constraint on our regions is that the read and write set of the optimization region must reside entirely in the cache.

Checking speculative compiler optimizations using multiple hardware threads has also been proposed. Three of these proposals are SlipStream [20], Master-Slave Speculative Parallelization (MSSP) [23], and FastForward [13]. They decompose an execution into two threads. The *leading* thread speculatively runs ahead and, with high probability, computes intermediate results of the program. The *trailing* thread (or threads in the case of MSSP) verifies and commits the leading thread's state. Speculative optimizations can be performed on the leading thread without requiring compensation code; the checks are performed by the trailing threads. In contrast, we do not require multiple cores per thread.

9. SUMMARY

This paper demonstrates that, through the introduction of architectural support for atomic regions, we can greatly facilitate the implementation of speculative optimizations in a compiler. This simplification occurs because atomic region abstraction permits the compiler to isolate the hot paths for the purpose of optimization. Infrequently-executed paths are replaced with simple checks (conditional aborts) that can be largely ignored during the optimization process. As a result, speculative optimizations can be performed without compensation code, enabling a great reduction in compiler complexity to achieve a given code quality.

Such hardware support is particularly compelling because the implementation complexity is modest. The atomicity abstraction is a clean one, placing few constraints on computer architects with regards to its implementation, especially as the compiler can effectively tolerate "best effort" implementations that do not guarantee forward progress in all circumstances. That said, the performance of the primitive is of fundamental importance, as any overhead or unnecessary serialization will negate much of the single-thread performance benefits enabled by this technique.

In light of the current industry-wide drive to improve efficiency, techniques that can improve performance while reducing power are particularly attractive. In our prototype compiler implementation, usage of atomic regions enabled average speedups of 12% across a suite of DaCapo benchmarks, with commensurate reductions in the number of micro-operation flowing through the pipeline. While these represent non-trivial gains in their own right, we believe that this does not represent the full potential of this approach.

10. ACKNOWLEDGMENTS

This research was supported in part by NSF CCR-0311340, NSF CAREER award CCR-03047260, and a gift from the Intel corporation. We would like to thank Haitham Akkary, Jesse Barnes, Mahesh Bhat, Alexander Ivanov, Konrad Lai, Egor Pasko (and the rest of the Apache Harmony developers), Pierre Salverda, Srikanth T. Srinivasan, and Yury Yudin for their technical support and feedback relating to this project.

11. REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

[2] S. M. Blackburn et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, Oct. 2006.

[3] Y. Chou, L. Spracklen, and S. G. Abraham. Store Memory-Level Parallelism Optimizations for Commercial Applications. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.

[4] A. Cristal et al. Out-of-Order Commit Processors. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, Feb. 2003.

[5] P. Damron et al. Hybrid Transactional Memory. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[6] J. C. Dehnert et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2003.

[7] B. Fahs et al. Performance Characterization of a Hardware Framework for Dynamic Optimization. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[8] G. Hamerly et al. SimPoint 3.0: Faster and More Flexible Program Analysis. *Journal of Instruction Level Parallelism*, 7, Sept 2005.

[9] Harmony Dynamic Runtime Layer Virtual Machine (DRLVM). http://harmony.apache.org/subcomponents/drlvm.

[10] W. M. Hwu et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1):229–248, Mar 1993.

[11] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 17th annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, 2002.

[12] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, Dec. 2006.

[13] L. ling Chen and Y. Wu. Fast Forward: Aggressive Compiler Optimization with Speculative Multi-Threaded Support. In *Workshop on Multithreaded Execution, Architecture and Compilation*, 2000.

[14] J. F. Martínez et al. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.

[15] R. Muth and S. Debray. Partial Inlining. Technical report, Univ. of Arizona, Dept. of Computer Science, 1997.

[16] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, 50(6):590–608, 2001.

[17] E. Perelman et al. Cross Binary Simulation Points. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2007.

[18] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[19] M. Smith. Overcoming the Challenges of Feedback-Directged Optiization. In *Proc. Proc. ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Jan. 2000.

[20] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[21] W. A. Wulf. Compilers and Computer Architecture. *IEEE Computer*, 14(7):41–47, 1981.

[22] C. Zilles and N. Neelakantam. Reactive Techniques for Controlling Software Speculation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[23] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.